El *kernel*

Josep Jorba Esteve

P07/M2103/02283



Índice

Introducción

El *kernel* del sistema GNU/Linux (al que habitualmente se le denomina Linux) [Vasb] es el corazón del sistema: se encarga de arrancar el sistema y, una vez éste ya es utilizable por las aplicaciones y los usuarios, gestiona los recursos de la máquina en forma de gestión de la memoria, sistema de ficheros, entrada/salida, procesos e intercomunicación de procesos.

Su origen se remonta al año 1991, cuando en agosto, un estudiante finlandés llamado Linus Torvalds anunció en una lista de *news* que había creado su propio núcleo de sistema operativo que funcionaba conjuntamente con software del proyecto GNU, y lo ofrecía a la comunidad de desarrolladores para que ésta lo probara y sugiriera mejoras para una mejor utilización. Así, se constituyó el origen del *kernel* del operativo, que más tarde se llamaría Linux.

Una de las particularidades de Linux es que, siguiendo la filosofía del Software Libre, se nos ofrece el código fuente del propio sistema operativo (del *kernel*), de manera que es una herramienta perfecta para la educación en temas de sistemas operativos.

Nota

El *kernel* Linux se remonta al año 1991, cuando Linus Torvalds lo puso a disposición de la comunidad. Es de los pocos operativos que, siendo ampliamente usado, se puede disponer de su código fuente.

La otra ventaja principal es que, al disponer de las fuentes, podemos recompilarlas para adaptarlas mejor a nuestro sistema, y podemos, asimismo, configurar sus parámetros para dar un mejor rendimiento al sistema.

En esta unidad veremos cómo manejar este proceso de preparación de un *kernel* para nuestro sistema: cómo, partiendo de las fuentes, podemos obtener una nueva versión del *kernel* adaptada a nuestro sistema. Del mismo modo, trataremos cómo se desarrolla la configuración y la posterior compilación y cómo realizar pruebas con el nuevo *kernel* obtenido.

1. El *kernel* del sistema GNU/Linux

El núcleo o *kernel* es la parte básica de cualquier sistema operativo [Tan87], y en él descansa el código de los servicios fundamentales para controlar el sistema entero. Básicamente, su estructura se puede separar en una serie de componentes de gestión orientados a:

- Gestión de procesos: qué tareas se van a ejecutar y en qué orden y con qué prioridad. Un aspecto importante es la planificación de CPU: ¿cómo se optimiza el tiempo de la CPU para ejecutar las tareas con el mayor rendimiento o interactividad posible con los usuarios?

- Intercomunicación de procesos y sincronización: ¿cómo se comunican tareas entre sí, con qué diferentes mecanismos y cómo pueden sincronizarse grupos de tareas?

- Gestión entrada/salida (E/S): control de periféricos y gestión de recursos asociados.

- Gestión de memoria: optimización del uso de la memoria, sistema de paginación y memoria virtual.

- Gestión de ficheros: cómo el sistema controla y organiza los ficheros presentes en el sistema y el acceso a los mismos.

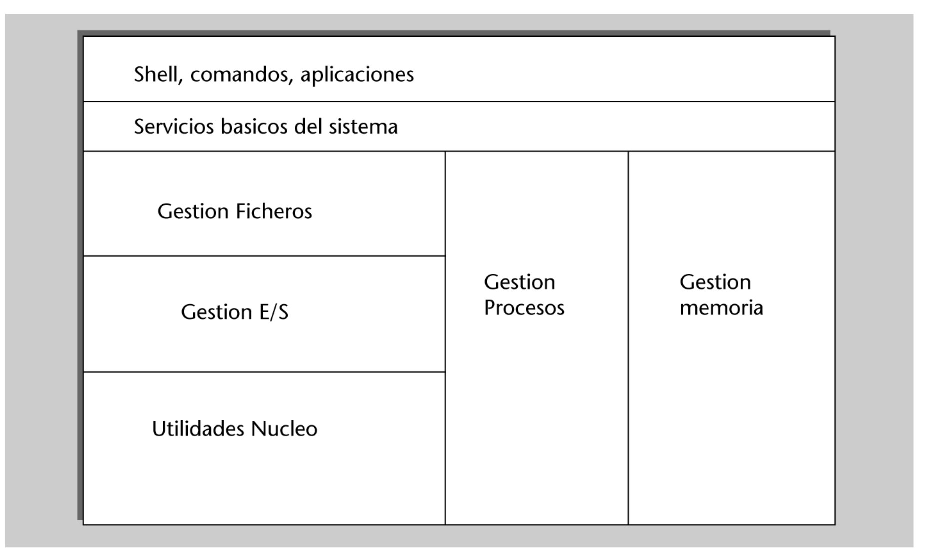


Figura 1. Funciones básicas de un *kernel* respecto de las aplicaciones y comandos ejecutados

En los sistemas propietarios, el *kernel* está perfectamente “oculto” bajo las capas del software del sistema operativo, y el usuario final no tiene una perspectiva clara de qué es el *kernel*, y no tiene ninguna posibilidad de cambiarlo u optimizarlo, si no es por el uso de esotéricos editores de “registros” internos,

o programas especializados de terceros (normalmente de alto coste). Además, el *kernel* suele ser único, es el que el fabricante proporciona, y éste se reserva el derecho de introducir las modificaciones que quiera y cuando quiera, así como tratar los errores que aparezcan en plazos no estipulados, mediante actualizaciones que nos ofrecen como "parches" de errores.

Uno de los principales problemas de esta aproximación es precisamente la disponibilidad de estos parches, disponer de las actualizaciones de los errores a su debido tiempo, y si son de seguridad, todavía más, ya que hasta que no estén corregidos no podemos garantizar la seguridad del sistema, para problemas ya conocidos. Muchas organizaciones, grandes empresas, gobiernos, instituciones científicas y militares no pueden depender de los caprichos de un fabricante para solucionar los problemas de sus aplicaciones críticas.

En este caso, el *kernel* Linux ofrece una solución de código abierto, con los consecuentes permisos de modificación, corrección, posibilidad de generación de nuevas versiones y actualizaciones de forma rápida, por parte de cualquiera que quiera y tenga los conocimientos adecuados para realizarlo.

Esto permite a los usuarios críticos controlar mejor sus aplicaciones y el propio sistema, y poder montar sistemas con el propio operativo "a la carta", personalizado al gusto de cada uno. Y disponer a su vez de un operativo con código abierto, desarrollado por una comunidad de programadores coordinados desde Internet, accesible ya sea para educación por disponer del código fuente y abundante documentación, o para la producción final de los sistemas GNU/Linux adaptados a necesidades individuales o de un determinado colectivo.

Al disponer de código fuente, se pueden aplicar mejoras y soluciones de forma inmediata, a diferencia del software propietario, donde debemos esperar las actualizaciones del fabricante. Podemos, además, personalizar el *kernel* tanto como necesitemos, requisito esencial, por ejemplo, en aplicaciones de alto rendimiento, críticas en el tiempo o soluciones con sistemas empotrados (como dispositivos móviles).

Siguiendo un poco de historia (rápida) del *kernel* [Kera] [Kerb]: lo comenzó a desarrollar un estudiante finlandés llamado Linus Torvalds, en 1991, con la intención de realizar una versión parecida a Minix [Tan87] (versión para PC de UNIX [Bac86]) para el procesador 386 de Intel. La primera versión publicada oficialmente fue la de Linux 1.0 en marzo de 1994, en la cual se incluía sólo la ejecución para la arquitectura i386 y soportaba máquinas de un sólo procesador. Linux 1.2 fue publicado en marzo de 1995, y fue la primera versión en dar cobertura a diferentes arquitecturas como Alpha, Sparc y Mips. Linux 2.0, en junio de 1996, añadió más arquitecturas y fue la primera versión en incorporar soporte multiprocesador (SMP) [Tum]. En Linux 2.2, enero de 1999, se incrementaron las prestaciones de SMP de manera significativa, y se añadieron controladores para gran cantidad de hardware. En la 2.4, en enero del

2001, se mejoró el soporte SMP, se incorporan nuevas arquitecturas soportadas y se integraron controladores para dispositivos USB, PC card (PCMCIA de los portátiles), parte de PnP (*plug and play*), soporte de RAID y volúmenes, etc. En la rama 2.6 del *kernel* (diciembre 2003), se mejora sensiblemente el soporte SMP, mejor respuesta del sistema de planificación de CPU, el uso de *threads* en el *kernel*, mejor soporte de arquitecturas de 64bits, soporte de virtualización, y mejor adaptación a dispositivos móviles.

Respecto al desarrollo, el *kernel*, desde su creación por Linus Torvalds en 1991 (versión 0.01), lo ha seguido manteniendo él mismo, pero a medida que su trabajo se lo permitía, y a medida que el *kernel* maduraba (y crecía), se ha visto ayudado a mantener las diferentes versiones estables del *kernel* por diferentes colaboradores, mientras Linus continuaba (en la medida de lo posible) desarrollando y recopilando aportaciones para la última versión de desarrollo del *kernel*. Los colaboradores principales en estas versiones han sido [lkm]:

Nota

El *kernel* tiene sus orígenes en el sistema MINIX, desarrollado por Andrew Tanenbaum, como un clon de UNIX para PC.

- 2.0 David Weinehall.
- 2.2 Alan Cox (también desarrolla y publica parches para la mayoría de versiones).
- 2.4 Marcelo Tosatti.
- 2.6 Andrew Morton / Linus Torvalds.

Para ver un poco la complejidad del *kernel* de Linux, veamos una tabla con un poco de su historia resumida en las diferentes versiones y su tamaño respectivo del código fuente. En la tabla se indican las versiones de producción únicamente; el tamaño (aproximado) está especificado en miles de líneas (K) de código fuente:

Versión	Fecha de la publicación	Líneas código (miles)
0.01	09-1991	10
1.0	03-1994	176
1.20	03-1995	311
2.0	06-1996	649
2.2	01-1999	1800
2.4	01-2001	3378
2.6	12-2003	5930

Como podemos comprobar, hemos pasado de unas diez mil líneas a seis millones.

En estos momentos el desarrollo continúa en la rama 2.6.x del *kernel*, la última versión estable, que incluye la mayoría de distribuciones como versión por defecto (algunas todavía incluyen algunas 2.4.x, pero 2.6.x suele ser una opción en la instalación); aunque cierto conocimiento de las anteriores versiones es imprescindible, ya que con facilidad podemos encontrar máquinas con distribuciones antiguas que no se hayan actualizado, a las que es posible que deba-

Nota

El kernel hoy en día ha alcanzado un grado de madurez y complejidad significativo.

mos mantener, o bien realizar un proceso de migración a versiones más actuales.

En la rama del 2.6, durante su desarrollo se aceleraron de forma significativa los trabajos del *kernel*, ya que tanto Linus Torvalds, como Andrew Morton (que mantienen Linux 2.6) se incorporaron (durante 2003) a OSDL (Open Source Developement Laboratory) [OSDa], un consorcio de empresas con el fin de promocionar el uso Open Source y GNU/Linux en la empresa (en el consorcio se encuentran entre otras muchas empresas con intereses en GNU/Linux: HP, IBM, Sun, Intel, Fujitsu, Hitachi, Toshiba, Red Hat, Suse, Transmeta...). En esos momentos se dio una situación interesante, ya que el consorcio OSDL hizo de patrocinador de los trabajos, tanto al mantenedor de la versión estable del *kernel* (Andrew) como la de desarrollo (Linus), trabajando a tiempo completo en las versiones, y en los temas relacionados. Linus se mantiene independiente, trabajando en el *kernel*, mientras Andrew se fue a trabajar a Google, donde continuaba a tiempo completo sus desarrollos, realizando parches con diferentes aportaciones al *kernel*. Recientemente, OSDL se reconvirtió en la fundación The Linux Foundation.

Nota

The Linux Foundation:
www.linux-foundation.org

Hay que tener en cuenta que con las versiones actuales del *kernel*, se ha alcanzado ya un alto grado de desarrollo y madurez, lo que hará que cada vez se amplíe más el tiempo entre la publicación de las versiones (no así de las revisiones parciales).

Además, otro factor a considerar es el tamaño y el número de personas que están trabajando en el desarrollo actual. En un principio había unas pocas personas que tenían un conocimiento global del *kernel* entero, mientras hoy en día tenemos muchas personas desarrollándolo, se cuenta que cerca de dos mil, con diferentes contribuciones, aunque el núcleo duro de desarrolladores se estima en unas pocas docenas.

Además, cabe tener en cuenta que la mayoría sólo tienen unos conocimientos parciales del *kernel*, y ni todos trabajan simultáneamente, ni su aportación es igual de relevante (algunas de ellas sólo corrigen errores sencillos); mientras que son unas pocas (como los mantenedores) las que disponen de un conocimiento total del *kernel*. Esto supone que se puedan alargar los desarrollos, y que las aportaciones se tengan que depurar, para ver que no entren en conflicto entre ellas, o decidir entre posibles alternativas de prestaciones para escoger.

Respecto a la numeración de las versiones del *kernel* de Linux ([lkm][DBo]), cabe tener en cuenta los aspectos siguientes:

a) Hasta la rama del *kernel* 2.6.x, las versiones del *kernel* Linux se regían por una división en dos series: una era la denominada "experimental" (la numerada impar en la segunda cifra, como 1.3.xx, 2.1.x o 2.5.x) y la otra es la de producción (serie par, como 1.2.xx, 2.0.xx, 2.2.x, 2.4.x y más). La serie expe-

rimental eran versiones que se movían rápidamente y se utilizaban para probar nuevas prestaciones, algoritmos o controladores de dispositivo, etc. Por la propia naturaleza de los *kernel*s experimentales, podían tener comportamientos impredecibles, como pérdidas de datos, bloqueos aleatorios de la máquina, etc. Por lo tanto, no debían utilizarse en máquinas destinadas a producción, a no ser que se quisiese probar una característica determinada (con los consecuentes peligros).

Los *kernel*s de producción (serie par) o estables eran *kernel*s con un conjunto de prestaciones bien definido, con un número bajo de errores conocidos y controladores de dispositivos probados. Se publicaban con menos frecuencia que los experimentales, y existían variedad de versiones, unas más buenas que otras. Los sistemas GNU/Linux se suelen basar en un determinado *kernel* estable elegido, no necesariamente el último *kernel* de producción publicado.

b) En la numeración actual del *kernel* Linux (utilizada en la rama 2.6.x), se siguen conservando algunos aspectos básicos: la versión viene indicada por unos números *X.Y.Z*, donde normalmente *X* es la versión principal, que representa los cambios importantes del núcleo; *Y* es la versión secundaria, y normalmente implica mejoras en las prestaciones del núcleo: *Y* es par en los núcleos estables e impar en los desarrollos o pruebas. Y *Z* es la versión de construcción, que indica el número de la revisión de *X.Y*, en cuanto a parches o correcciones hechas. Los distribuidores no suelen incluir la última versión del núcleo, sino la que ellos hayan probado con más frecuencia y puedan verificar que es estable para el software y componentes que ellos incluyen. Partiendo de este esquema de numeración clásico (que se siguió durante las ramas 2.4.x, hasta los inicios de la 2.6), tuvo algunas modificaciones para adaptarse al hecho de que el *kernel* (rama 2.6.x) se vuelve más estable (fijando X.Y a 2.6), y cada vez las revisiones son menores (por significar un salto de versión de los primeros números), pero el desarrollo continuo y frenético.

En los últimos esquemas, se llegan a introducir cuartos números, para especificar de Z cambios menores, o diferentes posibilidades de la revisión (con diferentes parches añadidos). La versión así definida con cuatro números es la que se considera estable (*stable)*. También son usados otros esquemas para las diversas versiones de test (normalmente no recomendables para entornos de producción), como sufijos -rc (*release candidate*), los -mm que son *kernel*s experimentales con pruebas de diferentes técnicas novedosas, o los -git que son una especie de "foto" diaria del desarrollo del *kernel*. Estos esquemas de numeración están en constante cambio para adaptarse a la forma de trabajar de la comunidad del *kernel*, y a sus necesidades para acelerar el desarrollo.

c) Para obtener el último *kernel* publicado, hay que acudir al archivo de *kernel*s Linux (en http://www.*kernel*.org) o al *mirror* local (en España http://www.es.*kernel*.org). También podrán encontrarse algunos parches al *kernel* original, que corrijan errores detectados a posteriori de la publicación del *kernel*.

Algunas de las características técnicas ([DBo][Arc]) del *kernel* Linux que podríamos destacar son:

- *Kernel* de tipo monolítico: básicamente es un gran programa creado como una unidad, pero conceptualmente dividido en varios componentes lógicos.

- Tiene soporte para carga/descarga de porciones del *kernel* bajo demanda, estas porciones se llaman módulos, y suelen ser características del *kernel* o controladores de dispositivo.

- *Threads* de *kernel*: para el funcionamiento interno se usan varios hilos (*threads*) de ejecución internos al *kernel*, que pueden estar asociados a un programa de usuario o bien a una funcionalidad interna del *kernel*. En Linux no se hacía un uso intensivo de este concepto. En las revisiones de la rama 2.6.x se ofreció mejor soporte, y gran parte del *kernel* se ejecuta usando diversos *threads* de ejecución.

- Soporte de aplicaciones *multithread*: soporte de aplicaciones de usuario de tipo *multithread*, ya que muchos paradigmas de computación de tipo cliente/servidor necesitan servidores capaces de atender múltiples peticiones simultáneas dedicando un hilo de ejecución a cada petición o grupo de ellas. Linux tiene una biblioteca propia de *threads* que puede usarse para las aplicaciones *multithread*, con las mejoras que se introdujeron en el *kernel*, también han permitido un mejor uso para implementar bibliotecas de *threads* para el desarrollo de aplicaciones.

- El *kernel* es de tipo no apropiativo (*nonpreemptive*): esto supone que dentro del *kernel* no pueden pararse llamadas a sistema (en modo supervisor) mientras se está resolviendo la tarea de sistema, y cuando ésta acaba, se reasume la ejecución de la tarea anterior. Por lo tanto, el *kernel* dentro de una llamada no puede ser interrumpido para atender otra tarea. Normalmente, los *kernel*s apropiativos están asociados a sistemas que trabajan en tiempo real, donde debe permitirse lo anterior para tratar eventos críticos. Hay algunas versiones especiales del *kernel* de Linux para tiempo real, que permiten esto por medio de la introducción de unos puntos fijos donde pueden intercambiarse. También se ha mejorado especialmente este concepto en la rama 2.6.x del *kernel*, permitiendo en algunos casos interrumpir algunas tareas del *kernel*, reasumibles, para tratar otras , resumiendo posteriormente su ejecución. Este concepto de *kernel* apropiativo también puede ser útil para mejorar tareas interactivas, ya que si se producen llamadas costosas al sistema, pueden provocar retardos en las aplicaciones interactivas.

- Soporte para multiprocesador, lo que llama multiprocesamiento simétrico (SMP). Este concepto suele englobar máquinas que van desde el caso simple de 2 hasta 64 CPU. Este tema se ha puesto de especial actualidad con

las arquitecturas de tipo *multicore,* permitiendo de 2 a 4 o más *cores* de CPU en máquinas accesibles a los usuarios domésticos. Linux puede usar múltiples procesadores, donde cada procesador puede manejar una o más tareas. Pero había algunas partes del *kernel* que disminuían el rendimiento, ya que están pensadas para una única CPU y obligan a parar el sistema entero en determinados bloqueos. SMP es una de las técnicas más estudiadas en la comunidad del *kernel* de Linux, y se han obtenido mejoras importantes en la rama 2.6. Ya del rendimiento SMP, depende en gran medida la adopción de Linux en los sistemas empresariales, en la faceta de sistema operativo para servidores.

- Sistemas de ficheros: el *kernel* tiene una buena arquitectura de los sistemas de ficheros, el trabajo interno se basa en una abstracción de un sistema virtual (VFS, *virtual file system*), que puede ser adaptada fácilmente a cualquier sistema real. Como resultado, Linux es quizás el operativo que más sistemas de ficheros soporta, desde el propio ext2, hasta msdos, vfat, ntfs, sistemas con *journal* como ext3, ReiserFS, JFS(IBM), XFS(Silicon), NTFS, iso9660 (CD), udf, etc. y se van añadiendo más en las diferentes revisiones.

Otras características menos técnicas (un poco de marketing):

a) Linux es gratis: junto con el software GNU, y el incluido en cualquier distribución, podemos tener un sistema UNIX completo prácticamente por el coste del hardware, y por la parte de los costes de distribución GNU/Linux, podemos obtenerla prácticamente gratis. Pero no está de más pagar por una distribución completa, con los manuales y apoyo técnico, a un coste menor comparado con lo que se paga por algunos sistemas propietarios, o contribuir con la compra al desarrollo de las distribuciones que más nos gusten, o nos sean prácticas.

b) Linux es personalizable: la licencia GPL nos permite leer y modificar el código fuente del *kernel* (siempre que tengamos los conocimientos adecuados).

c) Linux se ejecuta en hardware antiguo bastante limitado; es posible, por ejemplo, crear un servidor de red con un 386 con 4 MB de RAM (hay distribuciones especializadas en bajos recursos).

d) Linux es un sistema potente: el objetivo principal en Linux es la eficiencia, se intenta aprovechar al máximo el hardware disponible.

e) Alta calidad: los sistemas GNU/Linux son muy estables, bajo ratio de fallos, y reducen el tiempo dedicado a mantener los sistemas.

f) El *kernel* es bastante reducido y compacto: es posible colocarlo, junto con algunos programas fundamentales en un sólo disco de 1.44 MB (existen varias distribuciones de un sólo disquete con programas básicos).

g) Linux es compatible con una gran parte de los sistemas operativos, puede leer ficheros de prácticamente cualquier sistema de ficheros y puede comunicarse por red para ofrecer/recibir servicios de cualquiera de estos sistemas. Además, también con ciertas bibliotecas puede ejecutar programas de otros sistemas (como msdos, Windows, BSD, Xenix, etc.) en la arquitectura x86.

h) Linux está ampliamente soportado: no hay ningún otro sistema que tenga la rapidez y cantidad de parches y actualizaciones que Linux, ni en los sistemas propietarios. Para un problema determinado, hay infinidad de listas de correo y foros, que en pocas horas pueden permitir solucionar cualquier problema. El único problema está en los controladores de hardware reciente, que muchos fabricantes todavía se resisten a proporcionar si no es para sistemas propietarios. Pero esto está cambiando poco a poco, y varios de los fabricantes más importantes de sectores como tarjetas de vídeo (NVIDIA, ATI) e impresoras (Epson, HP,) comienzan ya a proporcionar los controladores para sus dispositivos.

2. Personalizar o actualizar el *kernel*

Como usuarios o administradores de sistemas GNU/Linux, tenemos que tener en cuenta las posibilidades que nos ofrece el *kernel* para adaptarlo a nuestras necesidades y equipos.

Normalmente, construimos nuestros sistemas GNU/Linux a partir de la instalación en nuestros equipos de alguna de las distribuciones de GNU/Linux, ya sean comerciales como Red Hat, Mandriva, Suse o "comunitarias" como Debian, Fedora.

Estas distribuciones aportan, en el momento de la instalación, una serie de *kernel*s Linux binarios ya preconfigurados y compilados, y normalmente tenemos que elegir qué *kernel* del conjunto de los disponibles se adapta mejor a nuestro hardware. Hay *kernel*s genéricos, o bien orientados a dispositivos IDE, otros a SCSI, otros que ofrecen una mezcla de controladores de dispositivos [AR01], etc.

Otra opción de instalación suele ser la versión del *kernel*. Normalmente las distribuciones usan una instalación que consideran lo suficientemente estable y probada como para que no cause problemas a los usuarios. Por ejemplo, a día de hoy muchas distribuciones vienen con versiones 2.6.x del *kernel* por defecto, ya que se considera la versión más estable del momento (en que salió la distribución). En algunos casos en el momento de la instalación puede ofrecerse la posibilidad de usar como alternativa, versiones más modernas, con mejor soporte para dispositivos más modernos (de última generación), pero quizás no tan probadas, en el momento que se publica la distribución.

Los distribuidores suelen, además, modificar el *kernel* para mejorar el comportamiento de su distribución o corregir errores que han detectado en el *kernel* en el momento de las pruebas. Otra técnica bastante común en las comerciales es deshabilitar prestaciones problemáticas, que pueden causar fallos a los usuarios, o necesitan una configuración especifica de la máquina, o bien una determinada prestación no se considera lo suficientemente estable para incluirla activada.

Nota

La posibilidad de actualizar y personalizar el *kernel* a medida ofrece buena adaptación a cualquier sistema, permitiendo una optimización y sintonización al mismo.

Esto nos lleva a considerar que, por muy bien que un distribuidor haga el trabajo de adaptar el *kernel* a su distribución, siempre nos podemos encontrar con una serie de problemas:

- El *kernel* no está actualizado en la última versión estable disponible; algunos dispositivos modernos no se soportan.
- El *kernel* estándar no dispone de soporte de los dispositivos que tenemos, porque no han estado habilitados.

- Los controladores que nos ofrece un fabricante necesitan una nueva versión del *kernel* o modificaciones.
- A la inversa, el *kernel* es demasiado moderno, tenemos hardware antiguo que ya no tiene soporte en los *kernel*s modernos.
- El *kernel*, tal como está, no obtiene las máximas prestaciones de nuestros dispositivos.
- Algunas aplicaciones que queremos usar requieren soporte de un *kernel* nuevo o de algunas de sus prestaciones.
- Queremos estar a la última, nos arriesgamos, instalando últimas versiones del *kernel* Linux.
- Nos gusta investigar o probar los nuevos avances del *kernel* o bien queremos tocar o modificar el *kernel*.
- Queremos programar un controlador para un dispositivo no soportado.
- ...

Por éstos y otros motivos podemos no estar contentos con el *kernel* que tenemos; se nos plantean entonces dos posibilidades: actualizar el *kernel* binario de la distribución o bien personalizarlo a partir de las fuentes.

Vamos a ver algunas cuestiones relacionadas con las diferentes opciones y qué suponen:

1) Actualización del *kernel* de la distribución: el distribuidor normalmente publica también las actualizaciones que van surgiendo del *kernel*. Cuando la comunidad Linux crea una nueva versión del *kernel*, cada distribuidor la une a su distribución y hace las pruebas pertinentes. Posteriormente al periodo de prueba, se identifican posibles errores, los corrige y produce la actualización del *kernel* pertinente respecto a la que ofrecía en los CD de la distribución. Los usuarios pueden descargar la nueva revisión de la distribución del sitio web, o bien actualizarla mediante algún sistema automático de paquetes vía repositorio de paquetes. Normalmente, se verifica qué versión tiene el sistema, se descarga el *kernel* nuevo y se hacen los cambios necesarios para que la siguiente vez el sistema funcione con el nuevo *kernel*, y se mantiene la versión antigua por si hay problemas.

> Este tipo de actualización nos simplifica mucho el proceso, pero no tiene por qué solucionar nuestros problemas, ya que puede ser que nuestro hardware no esté todavía soportado o la característica por probar del *kernel* no esté todavía en la versión que tenemos de la distribución; cabe recordar que no tiene por qué usar la última versión disponible (por ejemplo en *kernel*.org), sino aquella que el distribuidor considere estable para su distribución.

Si nuestro hardware tampoco viene habilitado por defecto en la nueva versión, estamos en la misma situación. O sencillamente, si queremos la última versión, este proceso no nos sirve.

2) Personalizar el *kernel* (proceso descrito con detalle en los apartados siguientes). En este caso, iremos a las fuentes del *kernel* y adaptaremos "a mano" el hardware o las características deseadas. Pasaremos por un proceso de configuración y compilación de las fuentes del *kernel* para, finalmente, crear un *kernel* binario que instalaremos en el sistema, y tenerlo, así, disponible en el siguiente arranque del sistema.

También aquí podemos encontrarnos con dos opciones más, o bien por defecto obtenemos la versión "oficial" del *kernel* (*kernel*.org), o bien podemos acudir a las fuentes proporcionadas por la propia distribución. Hay que tener en cuenta que distribuciones como Debian y Fedora hacen un trabajo importante de adecuación del *kernel*, y corrección de errores del *kernel* que afectan a su distribución, con lo cual podemos, en algunos casos, disponer de correcciones adicionales al código original del *kernel*. Otra vez más las fuentes ofrecidas por la distribución no tienen por qué corresponder a la última versión publicada.

Este sistema nos permite la máxima fiabilidad y control, pero a un coste de administración alto; ya que debemos disponer de conocimientos amplios de los dispositivos y de las características que estamos escogiendo (qué significan y qué implicaciones pueden tener), así como de las consecuencias que puedan tener las decisiones que tomemos.

3. Proceso de configuración y compilación

La personalización del *kernel* [Vasb] es un proceso costoso y necesita amplios conocimientos de lo que se está haciendo, y además, es una de las tareas críticas, de la cual depende la estabilidad del sistema, por la propia naturaleza del *kernel*, puesto que es el elemento central del sistema.

Nota

El proceso de obtención de un nuevo *kernel* personalizado pasa por obtener las fuentes, adaptar la configuración, compilar e instalar el kernel obtenido en el sistema.

Cualquier error de procedimiento puede comportar la inestabilidad o la pérdida del sistema. Por lo tanto, no está de más llevar a cabo cualquier tarea de *backup* de los datos de usuarios, datos de configuraciones que hayamos personalizado o, si disponemos de dispositivos adecuados, el *backup* completo del sistema. También es recomendable disponer de algún disquete de arranque (o distribución LiveCD con herramientas) que nos sirva de ayuda por si surgen problemas, o bien un disquete de rescate (*rescue disk*) que la mayoría de distribuciones permiten crear desde los CD de la distribución (o directamente proporcionan como CD de rescate para la distribución).

Sin ánimo de exagerar, casi nunca aparecen problemas si se siguen los pasos adecuadamente, se tiene conciencia de lo que se está haciendo y se toman algunas precauciones.

Vamos a ver el proceso necesario para instalar y configurar un *kernel* Linux. En los apartados siguientes, examinamos:

1) El caso de las versiones antiguas 2.4.x.

2) Algunas consideraciones sobre la migración a las 2.6.x

3) Detalles específicos de las versiones 2.6.x.

4) Un caso particular para la distribución Debian, que dispone de un sistema propio (*debian way*) de compilación más flexible.

Las versiones 2.4.x prácticamente ya no son ofrecidas por las distribuciones actuales, pero debemos considerar que en más de una ocasión nos veremos obligados a migrar un determinado sistema a nuevas versiones, o bien a mantenerlo en las antiguas, debido a incompatibilidades o existencia de hardware antiguo no soportado.

Los conceptos generales del proceso de compilación y configuración se explicarán en el primer apartado (2.4.x), ya que la mayoría de ellos son genéricos, y observaremos posteriormente las diferencias respecto las nuevas versiones.

3.1. Compilación *kernel* versiones 2.4.x

Las instrucciones son específicas para la arquitectura x86 Intel, mediante usuario *root* (aunque parte del proceso puede hacerse como usuario normal):

1) Obtener el *kernel*: por ejemplo, podemos acudir a www.kernel.org (o su servidor ftp) y descargar la versión que queramos probar. Hay *mirrors* para diferentes países, podemos, por ejemplo, acudir a www.es.kernel.org. En la mayoría de las distribuciones de GNU/Linux, como Fedora/Red Hat o Debian, también se ofrece como paquete el código fuente del *kernel* (normalmente con algunas modificaciones incluidas), si se trata de la versión del *kernel* que necesitamos, quizás sea preferible usar éstas (mediante los paquetes *kernel-source* o similares). Si queremos los últimos *kernel*s, quizás no estén disponibles en la distribución y tendremos que ir a *kernel.org*.

2) Desempaquetar el *kernel*: las fuentes del *kernel* solían colocarse y desempaquetarse sobre el directorio */usr/src*, aunque se recomienda utilizar algún directorio aparte para no mezclar con ficheros fuente que pueda traer la distribución. Por ejemplo, si las fuentes venían en un fichero comprimido de tipo bzip2:

```
bzip2 -dc linux-2.4.0.tar.bz2 | tar xvf -
```

Si las fuentes venían en un fichero gz, reemplazamos bzip2 por gzip. Al descomprimir las fuentes, se habrá generado un directorio linux-version_kernel, donde entraremos para establecer la configuración del *kernel*.

Antes de comenzar los pasos previos a la compilación, debemos asegurarnos de disponer de las herramientas correctas, en especial del compilador gcc, make y otras utilidades gnu complementarias en el proceso. Un ejemplo son las *modutils,* que disponen las diferentes utilidades para el uso y gestión de los módulos de *kernel* dinámicos. Asimismo, para las diferentes opciones de configuración hay que tener en cuenta una serie de prerrequisitos en forma de librerías asociadas a la interfaz de configuración usada (por ejemplo las ncurses para la interfaz menuconfig).

Se recomienda, en general, consultar la documentación del *kernel* (ya sea vía paquete, o en el directorio raíz de las fuentes) para conocer qué prerrequisitos, así como versiones de éstos, son necesarios para el proceso. Se recomienda examinar los ficheros *README* en el directorio "raíz", y el Documentation/Changes, o el índice de documentación del *kernel* en Documentation/00-INDEX.

Si hemos realizado anteriores compilaciones en el mismo directorio, deberemos asegurar que el directorio utilizado esté limpio de compilaciones anteriores; podemos limpiarlo con *make mrproper* (realizado desde el directorio "raíz").

Para el proceso de configuración del *kernel* [Vasb], tenemos varios métodos alternativos, que nos presentan interfaces diferentes para ajustar los múltiples

parámetros del *kernel* (que suelen almacenarse en un fichero de configuración, normalmente *.config* en el directorio "raíz" de las fuentes). Las diferentes alternativas son:

- **make config**: desde la línea de comandos se nos pregunta por cada opción, y se nos pide confirmación (y/n) –si deseamos o no, la opción o se nos piden los valores necesarios. O bien la configuración larga, en la que se nos piden muchas respuestas, y dependiendo de la versión, igual tenemos que responder a casi un centenar de preguntas (o más dependiendo de la versión).

- **make oldconfig**: sirve por si queremos reutilizar una configuración ya usada (normalmente almacenada en un fichero *.config*, en el directorio "raíz" de las fuentes), hay que tener en cuenta que sólo es válida si estamos compilando la misma versión del *kernel*, ya que diferentes versiones del *kernel* pueden variar en sus opciones.

- **make menuconfig**: configuración basada en menús textuales, bastante cómoda; podemos habilitar o inhabilitar lo que queramos y es más rápida que el *make config*.

- **make xconfig**: la más cómoda, basada en diálogos gráficos en X Window. Es necesario tener instaladas las bibliotecas de tcl/tk, ya que esta configuración está programada en este lenguaje. La configuración se basa en cuadros de diálogos y botones / casillas de activación, se hace bastante rápida y dispone de ayuda con comentarios de muchas de las opciones. Pero hay un defecto, puede ser que algunas de las opciones no aparezcan (depende de que el programa de configuración esté actualizado, y a veces no lo está). En este último caso, el *make config* (o el *menuconfig*) es el único que asegura disponer de todas las opciones elegibles; en los otros tipos de configuración depende de que se hayan podido adaptar a tiempo los programas a las nuevas opciones cuando se libera el *kernel*. Aunque en general se intentan mantener de forma equivalente.

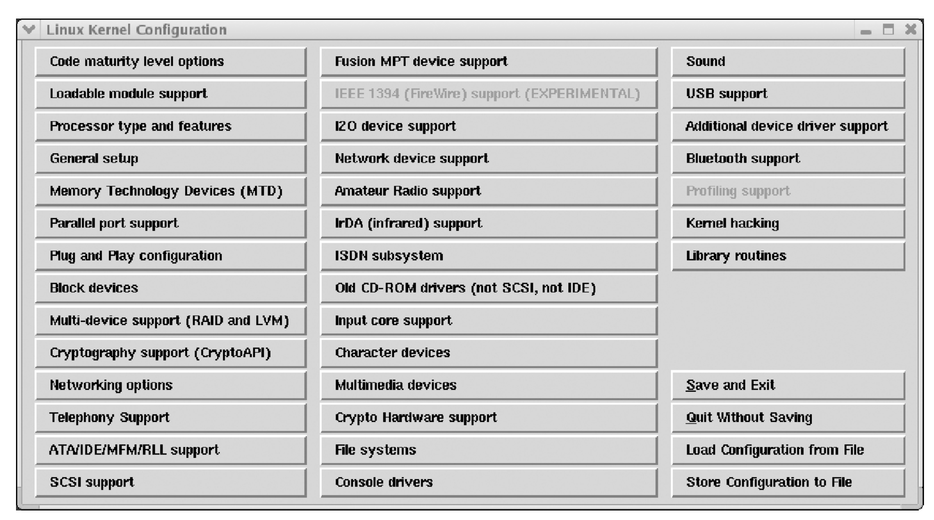


Figura 2. Configuración del *kernel* (make xconfig) desde interfaz gráfica en X Window

Una vez se haya hecho el proceso de configuración, hay que guardar el fichero (*.config),* ya que la configuración consume un tiempo importante. Además, puede ser de utilidad disponer de la configuración realizada si está planeado hacerla en varias máquinas parecidas o idénticas.

Otro tema importante en las opciones de configuración es que en muchos casos se nos va a preguntar por si una determinada característica la queremos integrada en el *kernel* o como módulo (en el apartado dedicado a los módulos daremos más detalles sobre los mismos). Ésta es una decisión más o menos importante, ya que el rendimiento del *kernel* (y por lo tanto, del sistema entero) en algunos casos puede depender de nuestra elección.

El *kernel* de Linux ha comenzado a ser de gran tamaño, tanto por complejidad como por los controladores (*drivers*) de dispositivo [AR01] que incluye. Si lo integrásemos todo, se podría crear un fichero del *kernel* bastante grande y ocupar mucha memoria, ralentizando, de ese modo, algunos aspectos de funcionamiento. Los módulos del *kernel* [Hen] son un método que permite separar parte del *kernel* en pequeños trozos, que serán cargados bajo demanda dinámicamente, cuando o bien por carga explícita o por uso de la característica sean necesarios.

La elección más normal es integrar lo que se considere básico para el funcionamiento o crítico en rendimiento dentro del *kernel*. Y aquellas partes o controladores de los que se vaya a hacer un uso esporádico o que se necesite dejarlos como módulos para futuras ampliaciones del equipo.

- Un caso claro son los controladores de dispositivo: si estamos actualizando la máquina, puede ser que a la hora de crear el *kernel* no conozcamos con seguridad qué hardware va a tener: por ejemplo, qué tarjeta de red; pero sí que sabemos que estará conectada a red, entonces, el soporte de red estará integrado en el *kernel*, pero en los controladores de las tarjetas podremos seleccionar unos cuantos (o todos) y ponerlos como módulos. Así, cuando tengamos la tarjeta podremos cargar el módulo necesario, o si después tenemos que cambiar una tarjeta por otra, sólo tendremos que cambiar el módulo que se va a cargar. Si hubiese sólo un controlador integrado en el *kernel* y cambiamos la tarjeta, esto obligaría a reconfigurar y recompilar el *kernel* con el controlador de la tarjeta nueva.

- Otro caso que suele aparecer (aunque no es muy común) es cuando necesitamos dos dispositivos que son incompatibles entre sí, o está funcionando uno o el otro (esto puede pasar, por ejemplo, con la impresora con cable paralelo y hardware que se conecta al puerto paralelo). Por lo tanto, en este caso tenemos que colocar como módulos los controladores y cargar o descargar el que sea necesario.

- Otro ejemplo podrían conformarlo los sistemas de ficheros (*filesystems*) normalmente esperaremos que nuestro sistema tenga acceso a algunos de

ellos, por ejemplo ext2 o ext3 (propios de Linux), vfat (de los Windows 95/98/ME), y los daremos de alta en la configuración del *kernel*. Si en otro momento tuviéramos que leer otro tipo no esperado, por ejemplo, datos guardados en un disco o partición de sistema NTFS de Windows NT/XP, no podríamos: el *kernel* no sabría o no tendría soporte para hacerlo. Si tenemos previsto que en algún momento (pero no habitualmente) se tenga que acceder a estos sistemas, podemos dejar los demás *filesystems* como módulos.

1) Compilación del *kernel*

Mediante el *make* comenzaremos la compilación, primero hay que generar las posibles dependencias entre el código, y luego el tipo de imagen de *kernel* que se quiere (en este caso una imagen comprimida, que suele ser la normal):

```
make dep
make bzImage
```

Cuando este proceso acabe, tenemos la parte integrada del *kernel*; nos faltan las partes que hayamos puesto como módulos:

```
make modules
```

Hasta este momento hemos hecho la configuración y compilación del *kernel*. Esta parte podía hacerse desde un usuario normal o bien desde el *root*, pero ahora necesitaremos forzosamente un usuario *root*, porque pasaremos a la parte de la instalación.

2) Instalación

Comenzamos instalando los módulos:

```
make modules_install
```

y la instalación del nuevo *kernel* (desde el directorio /usr/src/linux-version o el que hayamos utilizado como temporal):

```
cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.0
cp System.map /boot/System.map-2.4.0
```

el archivo bzImage es el *kernel* recién compilado, que se coloca en el directorio /boot. Normalmente, el *kernel* antiguo se encontrará en el mismo directorio /boot con el nombre vmlinuz, o bien vmlinuz-version-anterior y vmlinuz como un enlace simbólico al *kernel* viejo. Una vez tengamos nuestro *kernel*, es mejor conservar el antiguo, por si se producen fallos o mal funcionamiento del nuevo, poder recuperar el viejo. El fichero System.map contiene los símbolos disponibles en el *kernel* y es necesario para el proceso de arranque del mismo; también se coloca en el mismo directorio.

En este punto también hay que tener en cuenta que en el arranque el *kernel* puede necesitar la creación de ficheros tipo initrd, que sirve de imagen compuesta de algunos *drivers* básicos y se utiliza durante la carga del sistema, como primera fase, si el sistema necesita de esos *drivers* antes de proceder al arranque de algunos componentes. En algunos casos es imprescindible, debido a que, para poder cargar el resto del sistema, son necesarios controladores que deberán cargarse en una primera fase; un ejemplo de ello serían controladores específicos de disco como controladores RAID o de volúmenes, que serían necesarios para después, en una segunda fase, acceder al disco para cargar el resto del sistema.

El *kernel* puede generarse con o sin imagen initrd, dependerá de las necesidades del hardware o sistema concreto. En algunos casos la distribución impone la necesidad de usar imagen initrd, en otros casos dependerá de nuestro hardware. También se suele utilizar para vigilar el tamaño del *kernel*, de manera que puede cargarse lo básico de éste a través de la imagen initrd, y posteriormente cargar el resto en forma de módulos en una segunda fase. En el caso de necesitar la imagen initrd, se crearía con la utilidad mkinitrd (ver man, o taller del capítulo), dentro del directorio /boot.

3) El siguiente paso es decirle al sistema con qué *kernel* tiene que arrancar, aunque esto depende del sistema de arranque de Linux:

a) Desde arranque con lilo [Zan][Skoa], ya sea en el MBR (*master boot record*) o desde partición propia, hay que añadir al fichero de configuración (en: /etc/lilo.conf), por ejemplo las líneas:

```
image = /boot/vmlinuz-2.4.0
 label = 2.4.0
```

donde *image* es el *kernel* que se va arrancar, y *label* será el nombre con el que aparecerá la opción en el arranque. Podemos añadir estas líneas o modificar las que hubiera del *kernel* antiguo. Se recomienda añadirlas y dejar el *kernel* antiguo, por si aparecen problemas, poder recuperar el antiguo. En el fichero /etc/lilo.conf puede haber una o más configuraciones de arranque, tanto de Linux como de otros sistemas (como Windows).

Cada arranque se identifica por su línea *image* y el *label* que aparece en el menú de arranque. Hay una línea *default = label* donde se indica el *label* que se arranca por defecto. También podemos añadirle a las líneas anteriores un root = /dev/... para indicar la partición de disco donde está el sistema de archivos principal (el '/'), recordar que los discos tienen dispositivos como /dev/hda (1.r disco ide) /dev/hdb (2.º disco ide) o bien /dev/sdx para discos SCSI (o emulados), y la partición se indicaría como *root = /dev/hda2* si el '/' de nuestro Linux estuviese en la segunda partición del primer disco ide. Además con "*append* =" podemos añadir parámetros al arranque del *kernel*

[Gor]. Si el sistema utiliza, initrd, también habrá que indicar cuál es el fichero (que también estará localizado en /boot/initrd-versiónkernel), con la opción “initrd=”. Después de cambiar la configuración del lilo, hay que escribirla para que arranque:

```
/sbin/lilo -v
```

Reiniciamos (*reboot*) y arrancamos con el nuevo *kernel*.

Si tuviésemos problemas, podemos recuperar el antiguo *kernel*, escogiendo la opción del viejo *kernel*, y luego, mediante la acción de retocar el lilo.conf, podemos devolver la antigua configuración o estudiar el problema y reconfigurar y recompilar el *kernel* de nuevo.

b) Arranque con grub [Kan01][Pro]. El manejo en este caso es bastante simple, cabe añadir una nueva configuración formada por el *kernel* nuevo y sumarla como una opción más al fichero del grub. A continuación, reiniciar procediendo de forma parecida a la del lilo, pero recordando que en grub es suficiente con editar el fichero (típicamente /boot/grub/menu.lst) y reiniciar. También es mejor dejar la antigua configuración para poder recuperarse de posibles errores.

3.2. Migración a *kernel* 2.6.x

En el caso de tener que actualizar versiones de distribuciones antiguas, o bien realizar el cambio de generación del *kernel* mediante las fuentes, habrá que tomar una serie de consideraciones que tomar en cuenta, debido a las novedades introducidas en la rama 2.6.x del *kernel*.

Listamos algunos puntos concretos que cabe observar:

- Algunos de los módulos del *kernel* han cambiado de nombre, y algunos han podido desaparecer, hay que comprobar la situación de los módulos dinámicos que se cargan (por ejemplo, examinar /etc/modules y/o /etc/modules.conf), y editarlos para reflejar los cambios.

- Se han añadido nuevas opciones para la configuración inicial del *kernel*:como make gconfig, una interfaz basada en gtk (Gnome). Habrá que observar, como prerrequisito en este caso, las librerías de Gnome. La opción de make xconfig se ha implementado ahora con las librerías de qt (KDE).

- Se incrementan las versiones mínimas necesarias de varias utilidades necesarias para el proceso de compilación (consultar Documentation/Changes en las fuentes del *kernel*). En especial la versión del compilador gcc mínima.

- Ha cambiado el paquete por defecto para las utilidades de módulos, pasa a ser module-init-tools (en lugar de modutils usado en 2.4.x). Este paquete es un prerrequisito para la compilación de *kernel*s 2.6.x, ya que el cargador de módulos está basado en esta nueva versión.

- El sistema devfs, queda obsoleto en favor de udev, el sistema que controla el arranque (conexión) en caliente (*hotplug*) de dispositivos (y su reconocimiento inicial, de hecho simulando un arranque en caliente al iniciar el sistema), creando dinámicamente las entradas en el directorio /dev, sólo para los dispositivos que estén actualmente presentes.

- En Debian a partir de ciertas versiones del 2.6.x, para las imágenes binarias de *kernel*s, *headers* y código fuente cambia el nombre de los paquetes de kernel-images/source/headers a linux-image/source/headers.

- En algunos casos, los dispositivos de tecnologías nuevas (como SATA) pueden haber pasado de /dev/hdX a /dev/sdX. En estos casos habrá que editar las configuraciones de /etc/fstab y el *bootloader (*lilo o grub*)* para reflejar los cambios.

- Pueden existir algunos problemas con dispositivos de entrada/salida concretos. El cambio de nombres de módulos de *kernel* ha afectado entre otros a los dispositivos de ratón, lo que puede afectar asimismo a la ejecución de X-Window, a la verificación de qué modelos son necesarios y cargar los módulos correctos (por ejemplo el *psmouse*). Por otra parte, en el *kernel*, se integran los *drivers* de sonido Alsa. Si disponemos de los antiguos OSS, habrá que eliminarlos de la carga de módulos, ya que Alsa ya se encarga de la emulación de estos últimos.

- Respecto a las arquitecturas que soporta el *kernel*, hay que tener en cuenta que con los *kernel* 2.6.x, en las diferentes revisiones, se han ido incrementando las arquitecturas soportadas, lo que nos permitirá disponer de imágenes binarias del *kernel* en las distribuciones (o las opciones de compilación del *kernel*) más adecuadas para el soporte de nuestros procesadores. En concreto podemos encontrarnos con arquitecturas como i386 (para intel y AMD): soportando la compatibilidad de Intel en 32 bits para toda la familia de procesadores (en algunas distribuciones es usada 486 como arquitectura general), en algunas distribuciones se integran versiones diferenciadas para i686 (Intel a partir de pentium pro en adelante), para k7 (AMD Athlon en adelante), y las específicas de 64 bits, para AMD 64 bits, e Intel con extensiones em64t de 64 bits como los Xeon, y Multicores. Por otra parte, también existe la arquitectura IA64 para los modelos 64bits Intel Itanium. En la mayoría de los casos, las arquitecturas disponen de las capacidades SMP activadas en la imagen del *kernel* (a no ser que la distribución soporte versiones con SMP o sin, creadas independientemente, en este caso, suele añadirse el sufijo -smp a la imagen que lo soporta).

- En Debian, para la generación de imágenes inirtrd, a partir de ciertas versiones del *kernel* (>=2.6.12) se consideran obsoletas las mkinitrd-tools, que son substituidas por nuevas utilidades como initramfs-tools o yaird. Ambas permiten la construcción de la imagen initrd, siendo la primera la recomendada (por Debian).

3.3. Compilación de *kernel* versiones 2.6.x

En las versiones 2.6.x, teniendo en cuenta las consideraciones comentadas anteriormente, la compilación se desarrolla de forma semejante a la expuesta anteriormente:

Una vez descargado el *kernel* 2.6.x (sea x el número de revisión del *kernel*, o el par de números) en el directorio que se usará para la compilación, y comprobadas las versiones necesarias de las utilidades básicas, se procede al proceso de compilación, y limpieza de anteriores compilaciones:

```
# make clean mrproper
```

configuración de parámetros (recordar que si disponemos de un .config previo, nos permitirá no comenzar de cero la configuración). La configuración la realizamos a través de la opción escogida de *make* (dependiendo de la interfaz usada):

```
# make menuconfig
```

construcción de la imagen binaria del *kernel*

```
# make dep
# make bzImage
```

construcción de los módulos (aquellos especificados como tales):

```
# make modules
```

instalación de los módulos creados (/lib/modules/version)

```
# make modules_install
```

copia de la imagen a su posición final (suponiendo i386 como arquitectura):

```
# cp arch/i386/boot/bzimage /boot/vmlinuz-2.6.x.img
```

y finalmente, creación de la imagen initrd que consideremos necesaria, con las utilidades necesarias según la versión (ver comentario posterior). Y ajuste del *bootloader* lilo o grub según sea el utilizado.

Los últimos pasos (vmlinuz, system.map y initrd) de movimiento de archivos a /boot pueden realizarse también normalmente con el proceso:

```
# make install
```

pero hay que tener en cuenta que realiza todo el proceso, y actualizará los *bootloaders,* quitando antiguas configuraciones o alterándolas; asimismo, pueden alterarse los enlaces por defecto en el directorio /boot. Hay que tenerlo en cuenta a la hora de pensar en las configuraciones pasadas que queremos guardar.

Respecto a la creación del initrd, en Fedora/Red Hat éste se creara automáticamente con la opción "install". En Debian deberemos, o bien utilizar las técnicas del siguiente apartado, o bien crearlo expresamente con mkinitrd (versiones <=2.6.12) o, posteriormente, con mkinitramfs, o una utilidad denominada update-initramfs, especificando la versión del *kernel* (se asume que éste se llama vmlinuz-version dentro del directorio /boot):

```
# update-initramfs -c -k 'version'
```

3.4. Compilación en Debian del *kernel* (Debian way)

En Debian, además de los métodos examinados hay que añadir la configuración por el método denominado Debian Way. Un método que nos permite construir el *kernel* de una forma flexible y rápida.

Para el proceso nos serán necesarias una serie de utilidades (instalar los paquetes, o similares): kernel-package, ncurses-dev, fakeroot, wget, bzip2.

Podemos observar el método desde dos perspectivas, reconstruir un *kernel* equivalente al proporcionado por la distribución, o bien personalizarlo y utilizar así el método para construir un *kernel* equivalente personalizado.

En el primer caso, pasamos por obtener la versión de las fuentes del *kernel* que nos proporciona la distribución (sea x la revisión del *kernel* 2.6):

Nota

Puede verse el proceso "Debian way" de forma detallada en: http://kernel-handbook.alioth.debian.org/

```
# apt-get install linux-source-2.6.x
$ tar -xvjf /usr/src/linux-source-2.6.x.tar.bz2
```

donde obtenemos las fuentes y las descomprimimos (el paquete deja el archivo en /usr/src).

Instalación de herramientas básicas:

```
# apt-get install build-essential fakeroot
```

Comprobar dependencias de las fuentes

```
# apt-get build-dep linux-source-2.6.x
```

Y construcción del binario, según configuración preestablecida del paquete (semejante a la incluida en los paquetes *image* oficiales del *kernel* en Debian):

```
$ cd linux-source-2.6.x
$ fakeroot debian/rules binary
```

Existen algunos procedimientos extra para la creación de *kernel*s en base a diferentes niveles de *patch* proporcionados por la distribución, y posibilidades de generar diferentes configuraciones finales (puede verse la referencia de la nota para complementar estos aspectos).

En el segundo caso, más habitual, cuando deseamos un *kernel* personalizado, deberemos realizar un proceso semejante a través de un paso de personalización típico (por ejemplo, mediante make menuconfig), los pasos:

Obtención y preparación del directorio (aquí obtenemos los paquetes de la distribución, pero es equivalente obteniendo las fuentes desde *kernel.org*):

```
# apt-get install linux-source-2.6.x
$ tar xjf /usr/src/linux-source-2.6.x.tar.bz2
$ cd linux-source-2.6.x
```

a continuación realizamos la configuración de parámetros, como siempre podemos basarnos en ficheros .config que hayamos utilizado anteriormente, para partir de una configuración conocida (para la personalización, también puede usarse cualquiera de los otros métodos, xconfig, gconfig...):

```
$ make menuconfig
```

construcción final del *kernel* dependiendo de initrd o no, sin initrd disponible (hay que tener cuidado con la versión utilizada; a partir de cierta versión del *kernel*, puede ser obligatorio el uso de imagen initrd):

```
$ make-kpkg clean
$ fakeroot make-kpkg --revision=custom.1.0 kernel_image
```

o bien si disponemos de initrd disponible (construido ya)

```
$ make-kpkg clean
$ fakeroot make-kpkg --initrd --revision=custom.1.0 kernel_image
```

El proceso finalizará con la obtención del paquete asociado a la imagen del *kernel*, que podremos finalmente instalar:

```
# dpkg -i ../linux-image-2.6.x_custom.1.0_i386.deb
```

Añadimos, también en este apartado, otra peculiaridad a tener en cuenta en Debian, que es la existencia de utilidades para añadir módulos dinámicos de *kernel* proporcionados por terceros. En particular la utilidad module-assistant permite automatizar todo este proceso a partir de las fuentes del módulo.

Necesitamos disponer de los *headers* del *kernel* instalado (paquete linux-headers-version) o bien de las fuentes que utilizamos en la compilación del *kernel*. A partir de aquí module-assistant puede utilizarse interactivamente, permitiendo seleccionar entre una amplia lista de módulos registrados previamente en la aplicación, y puede encargarse de descargar el módulo, compilarlo e instalarlo en el *kernel* existente.

También en la utilización desde línea de comandos, podemos simplemente especificar (m-a es equivalente a module-assistant):

```
# m-a prepare
# m-a auto-install nombre_modulo
```

Lo cual prepara el sistema para posibles dependencias, descarga fuentes del módulo, compila y, si no hay problemas, instala para el presente *kernel*. El nombre del módulo podemos observarlo de la lista interactiva de module-assistant.

4. Parchear el *kernel*

En algunos casos también puede ser habitual la aplicación de parches (*patch)* al *kernel* [lkm].

> Un fichero de parche (*patch file*) respecto al *kernel* de Linux es un fichero de texto ASCII que contiene las diferencias entre el código fuente original y el nuevo código, con información adicional de nombres de fichero y líneas de código. El programa *patch (*ver *man patch*) sirve para aplicarlo al árbol del código fuente del *kernel* (normalmente en /usr/src).

Los parches suelen necesitarse cuando un hardware especial necesita alguna modificación en el *kernel*, o se han detectado algunos *bugs* (errores) posteriores a alguna distribución amplia de una versión del *kernel*, o bien quiere añadirse una nueva prestación concreta. Para corregir el problema (o añadir la nueva prestación), se suele distribuir un parche en lugar de un nuevo *kernel* entero. Cuando ya existen varios de estos parches, se unen con diversas mejoras del *kernel* anterior para formar una nueva versión del *kernel*. En todo caso, si tenemos el hardware problemático, o el error afecta a la funcionalidad o a la estabilidad del sistema y no podemos esperar a la siguiente versión del *kernel*, será necesario aplicar el parche.

El parche se suele distribuir en un fichero comprimido tipo bz2 (bunzip2, aunque también puede encontrarse en gzip con extensión .gz), como por ejemplo podría ser:

```
patchxxxx-2.6.21-pversion.bz2
```

donde xxxx suele ser algún mensaje sobre el tipo o finalidad del parche. 2.6.21 sería la versión del *kernel* al cual se le va a aplicar el parche, y pversion haría referencia a la versión del parche, del que también pueden existir varias. Hay que tener en cuenta que estamos hablando de aplicar parches a las fuentes del *kernel* (normalmente instaladas, como vimos, en /usr/src/linux o directorio similar).

Una vez dispongamos del parche, tendremos que aplicarlo, veremos el proceso a seguir en algún fichero readme que acompañe al parche, pero generalmente el proceso sigue los pasos (una vez comprobados los requisitos previos) de descomprimir el parche en el directorio de los ficheros fuente y aplicarlo sobre las fuentes del *kernel*, como por ejemplo:

```
cd /usr/src/linux (o /usr/src/linux-2.6.21 o la versión que sea).
```

```
bunzip2 patch-xxxxx-2.6.21-version.bz2
patch -p1 < patch-xxxxx-2.6.21-version
```

y posteriormente, tendremos que recompilar el *kernel* para volverlo a generar.

Los parches pueden obtenerse de diferentes lugares. Lo más normal es encontrarlos en el sitio de almacén de los *kernel*s (www.kernel.org) o bien en www.linuxhq.com, que tiene un archivo completo de ellos. En determinadas comunidades Linux (o usuarios individuales) también suelen ofrecer algunas correcciones, pero es mejor buscar en los sitios estándar para asegurar un mínimo de confianza en estos parches, y evitar problemas de seguridad con posibles parches "piratas". Otra vía es el fabricante de hardware que puede ofrecer ciertas modificaciones del *kernel* (o de controladores) para que funcionen mejor sus dispositivos (un ejemplo conocido es NVIDIA y sus *drivers* Linux para sus tarjetas gráficas).

Por último, señalaremos que en las distribuciones de GNU/Linux, muchas de ellas (Fedora/Red Hat, Mandriva...) ya ofrecen *kernel*s parcheados por ellos mismos, y sistemas para actualizarlos (algunos incluso de forma automática, como en el caso de Fedora/Red Hat y Debian). Normalmente, en sistemas de producción es más recomendable seguir las actualizaciones del fabricante, aunque éste no ofrecerá necesariamente el último *kernel* publicado, sino el que crea más estable para su distribución, bajo pena de perder prestaciones de última generación, o alguna novedad en las técnicas incluidas en el *kernel*.

Nota

En sistemas que se quieran tener actualizados, por razones de test o de necesidad de las últimas prestaciones, siempre se puede acudir a www.kernel.org y obtener el kernel más moderno publicado.

5. Los módulos del *kernel*

El *kernel* es capaz de cargar dinámicamente porciones de código (módulos) bajo demanda [Hen], para complementar su funcionalidad (se dispone de esta posibilidad desde la versión 1.2 del *kernel*). Por ejemplo, los módulos pueden añadir soporte para un sistema de ficheros o para dispositivos hardware específicos. Cuando la funcionalidad proporcionada por el módulo no es necesaria, el módulo puede ser descargado, liberando memoria.

Normalmente, bajo demanda, el *kernel* identifica una característica no presente en el *kernel* en ese momento, contacta con un *thread* del *kernel* denominado kmod (en las versiones *kernel* 2.0.x el daemon era llamado kerneld*)*, éste ejecuta un comando modprobe para intentar cargar el módulo asociado, a partir o de una cadena con el nombre de módulo, o bien de un identificador genérico; esta información se consulta en el fichero /etc/modules.conf en forma de alias entre el nombre y el identificador.

A continuación se busca en

```
/lib/modules/version_kernel/modules.dep
```

para saber si hay dependencias con otros módulos. Finalmente, con el comando insmod se carga el módulo desde /lib/modules/version_kernel/ (el directorio estándar para los módulos), la version_kernel es la versión del *kernel* actual, se utiliza el comando uname -r para determinarla. Por lo tanto, los módulos en forma binaria están relacionados con una versión concreta del *kernel*, y suelen colocarse en /lib/modules/version-kernel.

Si hay que compilarlos, se tiene que disponer de las fuentes y/o *headers* de la versión del núcleo al cual está destinado.

Nota

Los módulos aportan una flexibilidad importante al sistema, permitiendo que se adapte a situaciones dinámicas.

Hay unas cuantas utilidades que nos permiten trabajar con módulos (suelen aparecer en un paquete software llamado modutils*,* que se reemplazó por module-init-tools para la gestión de módulos de la rama 2.6.x):

- **lsmod:** podemos ver los módulos cargados en el *kernel* (la información se obtiene del pseudofichero /proc/modules). Se listan los nombres, las dependencias con otros (en [ ]), el tamaño del módulo en bytes, y el contador de uso del módulo; esto permite descargarlo si la cuenta es cero.

Ejemplo

Algunos módulos en una Debian:

```
Module         Size      Used by     Tainted: P
agpgart        37.344         3      (autoclean)
```

```
apm           10.024        1      (autoclean)
parport_pc    23.304        1      (autoclean)
lp            6.816         0      (autoclean)
parport       25.992        1      [parport_pc lp]
snd           30.884        0
af_packet     13.448        1      (autoclean)
NVIDIA        1.539.872     10
es1371        27.116        1
soundcore     3.972         4      [snd es1371]
ac97_codec    10.9640       0      [es1371]
gameport      1.676         0      [es1371]
3c59x         26.960        1
```

b) modprobe: intenta la carga de un módulo y de sus dependencias.

c) insmod: carga un módulo determinado.

d) depmod: analiza dependencias entre módulos y crea fichero de dependencias.

e) rmmod: saca un módulo del *kernel*.

f) Otros comandos pueden ser utilizados para depuración o análisis de los módulos, como modinfo*:* lista algunas informaciones asociadas al módulo, o ksyms*,* (sólo en versiones 2.4.x) permite examinar los símbolos exportados por los módulos (también en /proc/ksyms).

Ya sea por el mismo *kernel*, o por el usuario manualmente con insmod*,* normalmente para la carga se especificará el nombre del módulo, y opcionalmente determinados parámetros. Por ejemplo, en el caso de que sean dispositivos, suele ser habitual especificar las direcciones de los puertos de E/S o bien los recursos de IRQ o DMA. Por ejemplo:

```
insmod soundx io = 0x320 irq = 5
```

6. Futuro del *kernel* y alternativas

Los avances en el *kernel* de Linux en determinados momentos fueron muy rápidos, pero actualmente, ya con una situación bastante estable con los *kernel*s de la serie 2.6.x, cada vez pasa más tiempo entre las versiones que van apareciendo, y en cierta manera esto es bastante positivo. Permite tener tiempo para corregir errores cometidos, ver aquellas ideas que no funcionaron bien y probar nuevas ideas, que, si resultan, se incluyen.

Comentaremos en este apartado algunas de las ideas de los últimos *kernel*s, y algunas que están previstas, para dar indicaciones de lo que será el futuro próximo en el desarrollo del *kernel*.

Nota

El *kernel* continúa evolucionando, incorporando las últimas novedades en soporte hardware y mejoras en las prestaciones.

En la anterior serie, serie 2.4.x [DBo], incluida en la mayoría de distribuciones actuales, se realizaron algunas aportaciones en:

- Cumplimiento de los estándares IEEE POSIX, esto permite que muchos de los programas existentes de UNIX pueden recompilarse y ejecutarse en Linux.

- Mejor soporte de dispositivos: PnP, USB, Puerto Paralelo, SCSI...

- Soporte para nuevos *filesystems*, como UDF (CD-ROM reescribibles como un disco). Otros sistemas con *journal*, como los Reiser de IBM o el ext3, éstos permiten tener un *log* (*journal*) de las modificaciones de los sistemas de ficheros, y así poder recuperarse de errores o tratamientos incorrectos de los ficheros.

- Soporte de memoria hasta 4 GB, en su día surgieron algunos problemas (con *kernel*s 1.2.x) que no soportaban más de 128 MB de memoria (en aquel tiempo era mucha memoria).

- Se mejoró la interfaz /proc. Éste es un pseudosistema de ficheros (el directorio /proc) que no existe realmente en disco, sino que es simplemente una forma de acceder a datos del *kernel* y del hardware de una manera organizada.

- Soporte del sonido en el *kernel*, se añadieron parcialmente los controladores Alsa que antes se configuraban por separado.

- Se incluyó soporte preliminar para el RAID software, y el gestor de volúmenes dinámicos LVM1.

En serie actual, la rama del *kernel* 2.6.x [Pra], dispuso de importantes avances respecto a la anterior (con las diferentes revisiones .x de la rama 2.6):

- Mejores prestaciones en SMP, importante para sistemas multiprocesadores muy utilizados en entornos empresariales y científicos.

- Mejoras en el planificador de CPU (*scheduler*).

- Mejoras en el soporte multithread para las aplicaciones de usuario. Se incorporan nuevos modelos de *threads* NGPT (IBM) y NPTL (Red Hat) (con el tiempo se consolidó finalmente la NPTL).

- Soporte para USB 2.0.

- Controladores Alsa de sonido incorporados en el *kernel*.

- Nuevas arquitecturas de CPU de 64 bits, se soportan AMD x86_64 (también conocida como amd64) y PowerPC 64, y IA64 (arquitectura de los Intel Itanium).

- Sistemas de ficheros con *journal*: JFS, JFS2 (IBM), y XFS (Silicon Graphics).

- Mejoras de prestaciones en E/S, y nuevos modelos de controladores unificados.

- Mejoras en implementación TCP/IP, y sistema NFSv4 (compartición sistema de ficheros por red con otros sistemas).

- Mejoras significativas para *kernel* apropiativo: permite que internamente el *kernel* gestione varias tareas que se pueden interrumpir entre ellas, imprescindible para implementar eficazmente sistemas de tiempo real.

- Suspensión del sistema y restauración después de reiniciar (por *kernel*).

- UML, User Mode Linux, una especie de máquina virtual de Linux sobre Linux que permite ver un Linux (en modo usuario) ejecutándose sobre una máquina virtual. Esto es ideal para la propia depuración, ya que se puede desarrollar y testear una versión de Linux sobre otro sistema, esto es útil tanto para el propio desarrollo del *kernel*, como para un análisis de seguridad del mismo.

- Técnicas de virtualización incluidas en el *kernel*: en las distribuciones se han ido incorporando diferentes técnicas de virtualización, que necesitan extensiones en el *kernel*; cabe destacar, por ejemplo, *kernel*s modificados para Xen, o Virtual Server (Vserver).

- Nueva versión del soporte de volúmenes LVM2.

- Nuevo pseudo sistema de ficheros /*sys,* destinado a incluir la información del sistema, y dispositivos que se irán migrando desde el sistema /*proc,* dejando este último con información relacionada con los procesos, y su desarrollo en ejecución.

- Módulo FUSE para implementar sistemas de ficheros en espacio de usuario.

En el futuro se tiene pensado mejorar los aspectos siguientes:

- Incremento de la tecnología de virtualización en el *kernel*, para soportar diferentes configuraciones de operativos, y diferentes tecnologías de virtualización, así como mejor soporte del hardware para virtualización incluido en los procesadores que surjan en las nuevas arquitecturas.

- El soporte de SMP (máquinas multiprocesador), de CPU de 64 bits (Itanium de Intel, y Opteron de AMD), el soporte de CPUs multiCore.

- La mejora de sistemas de ficheros para *clustering*, y sistemas distribuidos.

- Por el contrario, la mejora para *kernel*s más optimizados para dispositivos móviles (PDA, teléfonos...).

- Mejora en el cumplimiento de los estándar POSIX, etc.

- Mejora de la planificación de la CPU, aunque en la serie inicial de la rama 2.6.x se hicieron muchos avances en este aspecto, todavía hay bajo rendimiento en algunas situaciones, en particular en el uso de aplicaciones interactivas de escritorio, se están estudiando diferentes alternativas, para mejorar éste y otros aspectos.

Nota

POSIX:
www.UNIX-systems.org/

También, aunque se aparta de los sistemas Linux, la FSF (Free Software Foundation) y su proyecto GNU siguen trabajando en su proyecto de acabar un sistema operativo completo. Cabe recordar que el proyecto GNU tenía como principal objetivo conseguir un clon UNIX de software libre, y las utilidades GNU sólo son el software de sistema necesario. A partir de 1991, cuando Linus consigue conjuntar su *kernel* con algunas utilidades GNU, se dio un primer paso que ha acabado en los sistemas GNU/Linux actuales. Pero el proyecto GNU sigue trabajando en su idea de terminar el sistema completo. En este momento disponen ya de un núcleo en el que pueden correr sus utilidades GNU. A este núcleo se le denomina Hurd; y a un sistema construido con él se le conoce como GNU/Hurd. Ya existen algunas distribuciones de prueba, en concreto, una Debian GNU/Hurd.

Nota

El proyecto GNU:
http://www.gnu.org/gnu/thegnuproject.html

Hurd fue pensado como el núcleo para el sistema GNU sobre 1990 cuando comenzó su desarrollo, ya que entonces la mayor parte del software GNU estaba desarrollado, y sólo faltaba el *kernel*. Fue en 1991 cuando Linus combinó GNU con su *kernel* Linux y creó así el inicio de los sistemas GNU/Linux. Pero Hurd sigue desarrollándose. Las ideas de desarrollo en Hurd son más complejas, ya que Linux podría considerarse un diseño "conservador", que partía de ideas ya conocidas e implantadas.

Nota

GNU y Linux, por Richard Stallman:
http://www.gnu.org/gnu/linux-and-gnu.html

En concreto, Hurd estaba pensada como una colección de servidores implementados sobre un *microkernel* Mach [Vah96], el cual es un diseño de núcleo

tipo *microkernel* (a diferencia de Linux, que es de tipo monolítico) desarrollado por la Universidad de Carnegie Mellon y posteriormente por la de Utah. La idea base era modelar las funcionalidades del *kernel* de UNIX como servidores que se implementarían sobre un núcleo básico Mach. El desarrollo de Hurd se retrasó mientras se estaba acabando el diseño de Mach, y éste se publicó finalmente como software libre, que permitiría usarlo para desarrollar Hurd. Comentar en este punto la importancia de Mach, ya que muchos operativos se han basado en ideas extraídas de él; el más destacado es el MacOS X de Apple.

El desarrollo de Hurd se retrasó más por la complejidad interna, ya que existían varios servidores con diferentes tareas de tipo *multithread* (de ejecución de múltiples hilos), y la depuración era extremadamente difícil. Pero hoy en día, ya se dispone de las primeras versiones de producción, así como de versiones de prueba de distribución GNU/Hurd.

Puede que en un futuro no tan lejano coexistan sistemas GNU/Linux con GNU/Hurd, o incluso sea sustituido el núcleo Linux por el Hurd, si prosperan algunas demandas judiciales contra Linux (léase caso SCO frente a IBM), ya que sería una solución para evitar problemas posteriores. En todo caso, tanto los unos como los otros sistemas tienen un prometedor futuro por delante. El tiempo dirá hacia dónde se inclina la balanza.

7. Taller: configuración del *kernel* a las necesidades del usuario

En este apartado vamos a ver un pequeño taller interactivo para el proceso de actualización y configuración del *kernel* en el par de distribuciones utilizadas: Debian y Fedora.

Una primera cosa imprescindible, antes de comenzar, es conocer la versión actual que tenemos del *kernel* con uname -r, para poder determinar cuál es la versión siguiente que queremos actualizar o personalizar. Y otra es la de disponer de medios para arrancar nuestro sistema en caso de fallos: el conjunto de CD de la instalación, el disquete (o CD) de rescate (actualmente suele utilizarse el primer CD de la distribución), o alguna distribución en LiveCD que nos permita acceder al sistema de ficheros de la máquina, para rehacer configuraciones que hayan causado problemas. Además de hacer *backup* de nuestros datos o configuraciones importantes.

Veremos las siguientes posibilidades:

1) Actualización del *kernel* de la distribución. Caso automático de Debian.
2) Actualización automática en Fedora.
3) Personalización de un *kernel* genérico (tanto Debian como Fedora). En este último caso los pasos son básicamente los mismos que los que se presentan en el apartado de configuración, pero haremos algunos comentarios más.

7.1. Actualizar *kernel* en Debian

En el caso de la distribución Debian, la instalación puede hacerse también de forma automática, mediante el sistema de paquetes de APT. Puede hacerse tanto desde línea de comandos, como con gestores APT gráficos (synaptic, gnome-apt...).

Vamos a efectuar la instalación por línea de comandos con apt-get, suponiendo que el acceso a las fuentes apt (sobre todo a los Debian originales) está bien configurado en el fichero de /etc/apt/sources.list. Veamos los pasos:

1) Actualizar la lista de paquetes:

```
# apt-get update
```

2) Listar paquetes asociados a imágenes del *kernel*:

```
# apt-cache search linux-image
```

3) Elegir una versión adecuada a nuestra arquitectura (genérica, 386/486/686 para Intel, k6 o k7 para amd, o en particular para 64Bits versiones amd64, intel

y amd, o ia64, para Intel Itanium). La versión va acompañada de versión *kernel*, revisión de Debian del *kernel* y arquitectura. Por ejemplo: 2.6.21-4-k7, *kernel* para AMD Athlon, revisión Debian 4 del *kernel* 2.6.21.

4) Comprobar, para la versión elegida, que existan los módulos accesorios extras (con el mismo número de versión). Con apt-cache buscamos si existen otros módulos dinámicos que puedan ser interesantes para nuestro hardware, según la versión del *kernel* a instalar. Recordar que, como vimos en la “Debian way”, también existe la utilidad module-assistant, que nos permite automatizar este proceso después de la compilación del *kernel*. Si los módulos necesarios no fueran soportados, esto nos podría impedir actualizar el *kernel* si consideramos que el funcionamiento del hardware problemático es vital para el sistema.

5) Buscar, si queremos disponer también del código fuente del *kernel*, los Linux-source-version (sólo el 2.6.21, o sea, los números principales), y los *kernel-headers* correspondientes, por si más tarde queremos hacer un *kernel* personalizado: en este caso, el *kernel* genérico correspondiente parcheado por Debian.

6) Instalar lo que hayamos decidido; si queremos compilar desde las fuentes o simplemente disponer del código:

```
# apt-get install linux-image-version
# apt-get install xxxx-modules-version (si fuera necesarios
algunos módulos)
```

y

```
# apt-get install linux-source-version-generica
# apt-get install linux-headers-version
```

7) Instalar el nuevo *kernel*, por ejemplo en el *bootloader* lilo. Normalmente esto se hace automáticamente. Si se nos pregunta si tenemos el initrd activado, habrá que verificar el fichero de lilo (/etc/lilo.conf) e incluir en la configuración lilo de la imagen nueva la nueva línea:

initrd = /initrd.img-version (o /boot/initrd.img-version)

una vez hecha esta configuración, tendríamos que tener un lilo del modo (fragmento), suponiendo que initrd.img y vmlinuz sean enlaces a la posición de los ficheros del nuevo *kernel*:

```
default = Linux

image = /vmlinuz
      label = Linux
      initrd = /initrd.img
#     restricted
#     alias = 1
```

```
image = /vmlinuz.old
      label = LinuxOLD
      initrd = /initrd.img.old
#     restricted
#     alias = 2
```

Tenemos la primera imagen por defecto, la otra es el *kernel* antiguo. Así, desde el menú lilo podremos pedir una u otra, o simplemente cambiando el *default* recuperar la antigua. Siempre que realicemos cambios en /etc/lilo.conf no debemos olvidarnos de reescribirlos en el sector correspondiente con el comando /sbin/lilo o /sbin/lilo -v.

7.2. Actualizar *kernel* en Fedora/Red Hat

La actualización del *kernel* en la distribución Fedora/Red Hat es totalmente automática por medio de su servició de gestión de paquetes, o bien por los programas gráficos que incluye la distribución para la actualización; por ejemplo, en los Red Hat empresariales existe uno llamado up2date. Normalmente, lo encontraremos en la barra de tareas o en el menú de Herramientas de sistema de Fedora/Red Hat.

Este programa de actualización básicamente verifica los paquetes de la distribución actual frente a una base de datos de Fedora/Red Hat, y da la posibilidad de bajarse los paquetes actualizados, entre ellos los del *kernel*. Este servicio de Red Hat empresarial funciona por una cuenta de servicio, y Red Hat lo ofrece por pago. Con este tipo de utilidades la actualización del *kernel* es automática.

Por ejemplo, en la figura 10, observamos que una vez puesto en ejecución nos ha detectado una nueva versión del *kernel* disponible y podemos seleccionarla para que nos la descargue:

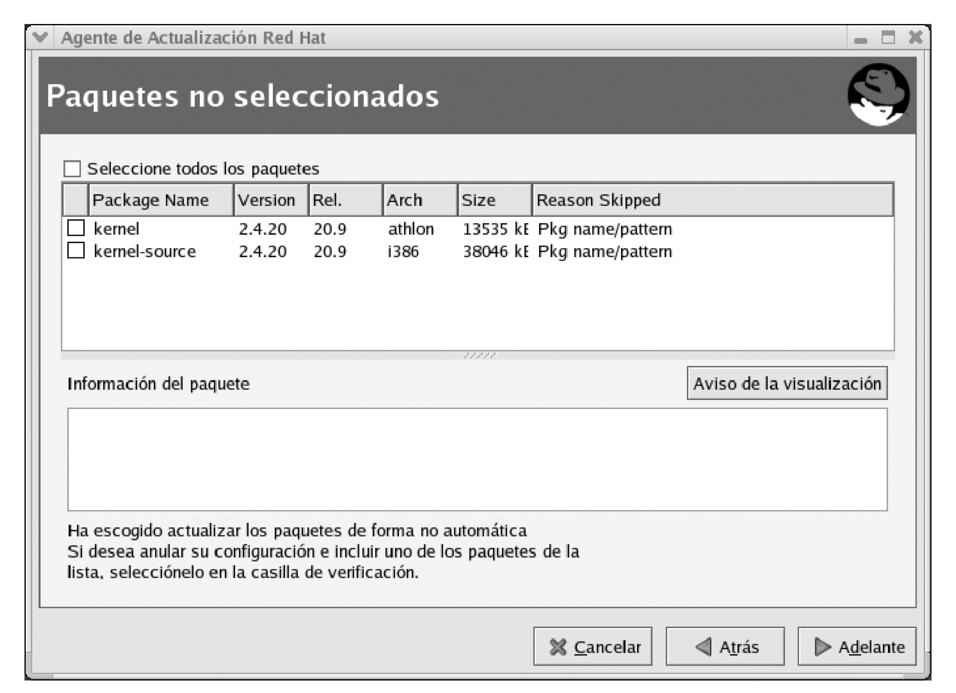


Figura 3. El servicio de actualización de Red Hat (Red Hat Network up2date) muestra la actualización del *kernel* disponible y sus fuentes.

En Fedora podemos o bien utilizar las herramientas gráficas equivalentes, o simplemente con yum directamente, si conocemos la disponibilidad de nuevos *kernels*:

```
# yum install kernel kernel-source
```

Una vez descargada, se procederá a su instalación, normalmente también de forma automática, ya dispongamos de grub o lilo, como gestores de arranque. En el caso de grub, suele ser automático y deja un par de entradas en el menú, una para la versión más nueva y otra para la antigua. Por ejemplo, en esta configuración de grub (el fichero está en /boot/grub/grub.conf o bien /boot/grub/menu.lst), tenemos dos *kernels* diferentes, con sus respectivos números de versión:

```
#fichero grub.conf
default = 1
timeout = 10
splashimage = (hd0,1)/boot/grub/splash.xpm.gz

title Linux (2.6.20-2945)
root (hd0,1)
kernel /boot/vmlinuz-2.6.20-2945 ro root = LABEL = /
initrd /boot/initrd-2.6.20-18.9.img

title LinuxOLD (2.6.20-2933)
root (hd0,1)
kernel /boot/vmlinuz-2.4.20-2933 ro root = LABEL = /
initrd /boot/initrd-2.4.20-2933.img
```

Cada configuración incluye un título, que aparecerá en el arranque; el *root*, o partición del disco desde donde arrancar; el directorio donde se encuentra el fichero correspondiente al *kernel*, y el fichero initrd correspondiente.

En el caso de que dispongamos de lilo (por defecto se usa grub) en la Fedora/Red Hat como gestor, el sistema también lo actualiza (fichero /etc/lilo.conf), pero luego habrá que reescribir el arranque con el comando /sbin/lilo manualmente.

Cabe señalar, asimismo, que con la instalación anterior teníamos posibilidades de bajarnos las fuentes del *kernel*; éstas, una vez instaladas, están en /usr/src/linux-version, y pueden configurarse y compilarse por el procedimiento habitual como si fuese un *kernel* genérico. Hay que mencionar que la empresa Red Hat lleva a cabo un gran trabajo de parches y correcciones para el *kernel* (usado después en Fedora), y que sus *kernel* son modificaciones al estándar genérico con bastantes añadidos, por lo cual puede ser mejor utilizar las fuentes propias de Red Hat, a no ser que queramos un *kernel* más nuevo o experimental que el que nos proporcionan.

7.3. Personalizar e instalar un *kernel* genérico

Vamos a ver el caso general de instalación de un *kernel* a partir de sus fuentes. Supongamos que tenemos unas fuentes ya instaladas en /usr/src (o el prefijo correspondiente). Normalmente, tendremos un directorio linux, o linux-version o sencillamente el número version; éste será el árbol de las fuentes del *kernel*.

Estas fuentes pueden provenir de la misma distribución (o que las hayamos bajado en una actualización previa), primero será interesante comprobar si son las últimas disponibles, como ya hemos hecho antes con Fedora o Debian. O si queremos tener las últimas y genéricas versiones, podemos ir a kernel.org y bajar la última versión disponible, mejor la estable que las experimentales, a no ser que estemos interesados en el desarrollo del *kernel*. Descargamos el archivo y descomprimimos en /usr/src (u otro elegido, casi mejor) las fuentes del *kernel*. También podríamos buscar si existen parches para el *kernel* y aplicarlos (según hemos visto en el apartado 4.4).

A continuación comentaremos los pasos que habrá que realizar; lo haremos brevemente, ya que muchos de ellos los tratamos anteriormente cuando trabajamos la configuración y personalización.

Nota

Sería conveniente releer el apartado 3.4.3.

1) Limpiar el directorio de pruebas anteriores (si es el caso):

```
make clean mrproper
```

2) Configurar el *kernel* con, por ejemplo: make menuconfig (o xconfig, gconfig o oldconfig). Lo vimos en el apartado 4.3.

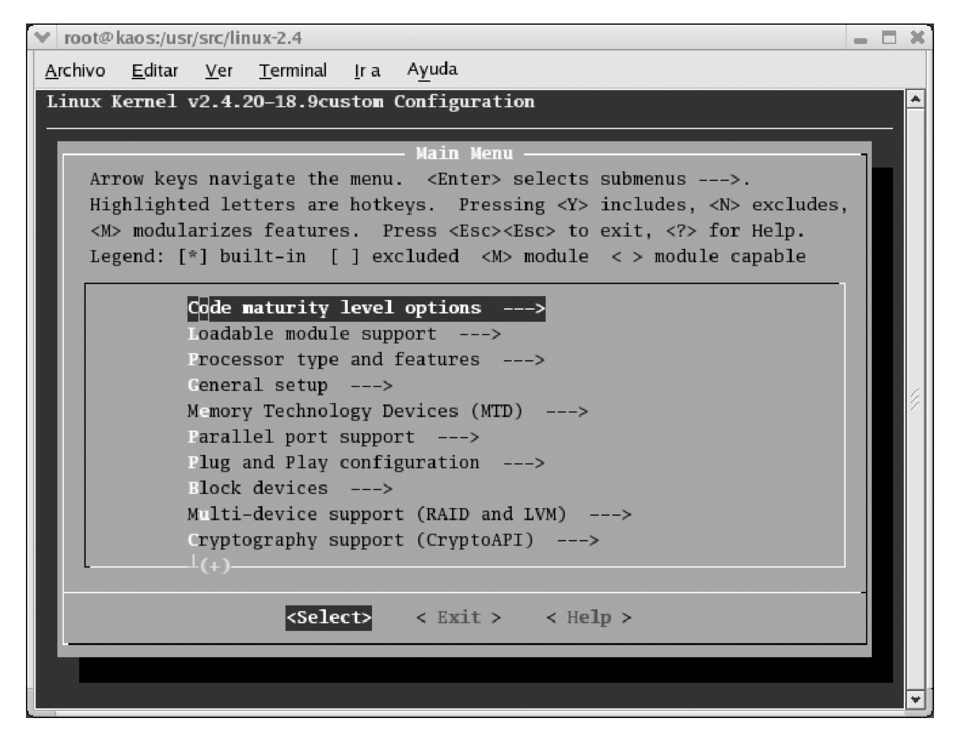


Figura 4. Configuración del *kernel* por menús textuales

4) Dependencias y limpieza de compilaciones anteriores:

```
make dep
```

5) Compilación y creación de la imagen del *kernel*: make bzImage. También sería posible zImage si la imagen fuera más pequeña, pero es más normal bzImage, que optimiza el proceso de carga y la compresión para *kernel*s más grandes. En algún hardware muy antiguo puede no funcionar y ser necesario zImage. El proceso puede durar desde unos pocos minutos a una hora en hardware moderno (CPU de 1-3 GHz) y horas en hardware muy antiguo. Cuando acaba, la imagen se halla en: /usr/src/directorio-fuentes/arch/i386/boot.

6) Ahora compilamos los módulos con *make modules*. Hasta este momento no hemos modificado nada en nuestro sistema. Ahora tendremos que proceder a la instalación.

7) En el caso de los módulos, si probamos alguna versión antigua del *kernel* (rama 2.2 o primeras de la 2.4), hay que tener cuidado, ya que en alguna se sobrescribían los antiguos (en las últimas 2.4.x o 2.6.x ya no es así).

Pero también cuidado si estamos compilando una versión que es la misma (exacta numeración) que la que tenemos (los módulos se sobrescribirán), mejor hacer un *backup* de los módulos:

```
cd /lib/modules
tar -cvzf old_modules.tgz versionkernel-antigua/
```

Así, tenemos una versión en .tgz que podríamos recuperar después en caso de problemas. Y, finalmente, instalamos los módulos con:

```
make modules install
```

8) Ahora podemos pasar a la instalación del *kernel*, por ejemplo con:

```
# cd /usr/src/directorio-Fuentes/arch/i386/boot
# cp bzImage /boot/vmlinuz-versionkernel
# cp System.map /boot/System.map-versionkernel
# ln -s /boot/vmlinuz-versionkernel /boot/vmlinuz
# ln -s /boot/System.map-versionkernel /boot/System.map
```

Así colocamos el fichero de símbolos del *kernel* (System.map) y la imagen del *kernel*.

9) Ya sólo nos queda poner la configuración necesaria en el fichero de configuración del gestor de arranque, ya sea lilo (/etc/lilo.conf) o grub (/boot/grub/grub.conf) según las configuraciones que ya vimos con Fedora o Debian. Y re-

cordar, en el caso de lilo, que habrá que volver a actualizar la configuración con /sbin/lilo o /sbin/lilo -v.

10) Reiniciar la máquina y observar los resultados (si todo ha ido bien).

Actividades

1) Determinar la versión actual del *kernel* Linux incorporada en nuestra distribución. Comprobar las actualizaciones disponibles de forma automática, ya sea en Debian *(apt)* o en Fedora/Red Hat (vía yum o up2date).

2) Efectuar una actualización automática de nuestra distribución. Comprobar posibles dependencias con otros módulos utilizados (ya sean pcmcia, u otros), y con el *bootloader* (lilo o grub) utilizado. Se recomienda *backup* de los datos importantes del sistema (cuentas de usuarios y ficheros de configuración modificados), o bien realizarlo en otro sistema del que se disponga para pruebas.

3) Para nuestra rama del *kernel*, determinar la última versión disponible (consultar www.kernel.org), y realizar una instalación manual con los pasos examinados en la unidad. La instalación final puede dejarse opcional, o bien poner una entrada dentro del *bootloader* para las pruebas del nuevo *kernel*.

4) En el caso de la distribución Debian, además de los pasos manuales, existe como vimos una forma especial (recomendada) de instalar el *kernel* a partir de sus fuentes mediante el paquete *kernel-package*.

Otras fuentes de referencia e información

[Kerb] Sitio que proporciona un almacén de las diversas versiones del *kernel* Linux y sus parches.

[Kera] [lkm] Sitios web que recogen una parte de la comunidad del *kernel* de Linux. Dispone de varios recursos de documentación y listas de correo de la evolución del *kernel*, su estabilidad y las nuevas prestaciones que se desarrollan.

[DBo] Libro sobre el *kernel* de Linux 2.4, que detalla los diferentes componentes y su implementación y diseño. Existe una primera edición sobre el *kernel* 2.2 y una nueva actualización al *kernel* 2.6.

[Pra] Artículo que describe algunas de las principales novedades de la nueva serie 2.6 del *kernel* Linux.

[Ker] [Mur] Proyectos de documentación del *kernel*, incompletos pero con material útil.

[Bac86] [Vah96] [Tan87] Algunos textos sobre los conceptos, diseño e implementación de los *kernel*s de diferentes versiones UNIX.

[Skoa][Zan01][Kan][Pro] Para tener más información sobre los cargadores lilo y grub.